reference

```
int var = 123;
int &rvar = var; //int &var = int& var
/* & erzeugt alias für einen Bezeichner.
=> var und rvar nun absolut identisch.
Referenzen finden ihren
Hauptverwendungszweck als Parameter und
Rückgabewerte von Funktionen */
```

raw <- pointer -> smart

```
int *iptr{nullptr}; //Deklaration
int ival = 123;
iptr = &ival; //iptr bekommt ival's Adresse
*iptr = 234; //ival = 234

if (ptr != nullptr)
  //pointer enthält gültige Adresse
```

```
/* Benötigen ALLE den namespace der Standard-
Bibliothek (auch: make_…) + Headerdatei
„memmory“ muss inkludiert werden */ // =>
#include <memory> //NICHT VERGESSEN!
using namespace std; //scope bevorzugen

//list: shared_ptr, weak_ptr, unique_ptr, auto_ptr
//Generische Definition:
std::…_ptr<(@d)> ptr = std::make_…<(@d)>((@p));
/* wobei (@d) der Datentyp, … der Name des
entsprechenden Smartprt und (@p) die vom
Konstruktor des Datentyps angeforderten
Parameter sind*/ // Alternativ auch:
std::…_ptr<(@d)> ptr(new (@d)((@p));
//Es gibt KEIN make_weak<>()!

//unique_ptr is NOT copyable:
std::unique_ptr<(@d)> ptr2 = ptr; //ERROR
//stattdessen:
std::unique_ptr<(@d)> ptr3(std::move(ptr));

unique_ptr /* Bessere alternative zu auto_ptr
(kann man z.B. in std Container packen). Er
teilt seinen Speicherbereich nicht. Gibt Re-
source frei, wenn Gültigkeitsber. verlassen*/

shared_ptr /* Teilt seinen Speicherbereich
=> mehrere Zeiger verweisen darauf. Enthält
use_count() - Zeiger Anzahl ermitteln
reset() - Zeiger Referenz danach: nullptr)*/
weak_ptr /*Unsichtbarer Zugriff auf
shared_prt (besetzt quasi keine Resource).
Kein gegenseitiges warten zweier shared_ptr*/
//Beispiele:
unique_ptr<Automat> a_ptr(new Automat{});
a_ptr->setdata(data);
unique_ptr<int> ptr = make_unique<int>(5);

/*WICHTIG: JEDES new sollte in einem smart
pointer verwendet werden! (man spart sich das
explizite freigeben von Speicher)*/
```

functions

```
//Funktion deklarieren:
int addieren(int, int); //Call-by-Value
int addieren(const int, const int); //C-b-V
int addieren(int&, int&);//C-b-Reference
int addieren(const int&, const int&);//C-b-R

//Default Parameter Werte festlegen
int addieren(int=0, int=0); /* Wenn benutzt,
dann ALLE param. rechts davon auch festlegen!*/
//Funktion definieren:
int addieren(const int val, const int va2) {
  return val + va2;
}
/* In der Regel konstante Referenzen als
Parameter und Rückgabewerte verwenden */
```

definit-, declaration

```
extern int ivar; //Deklaration
//└=> ohne Speicherplatz Reservierung
ivar {12345}; //Initialisierung (nach C++11)
/* Veralteter Stil: ivar = 12345;
Nur noch bei einfachen Datentypen üblich,
bei komplexen Typen “{…}“ bevorzugen! */

unsigned int ivar{0}; //Ohne Vorzeichen
const int ivar{0}; //Konstant

//Basis Datentypen:
bool /*logisch*/, auto /*Typinferenz*/
char < short < int < long < long long //Z
float < double < long double //Gleitkomma

//Dynamisch Objekte mit >>new<< anlegen:
int *iptr = new int{5}; //neues int-Objekt
int *iptr = new int[5]; //Dynamischer Array
```

casts

```
double pi{3.14};
//Implizit (durch Compiler)
int ival = pi;  //3,14 -> 3 (Verengung)
/*Um eine Verengung zu verbieten, C++11
einheitliche Initialisierung verwenden*/
int ival{pi};//ERROR: Narrowing (Verengung)

//Explizite Typumwandlung:
int ival = static_cast<int>(pi); //Syntax

static_cast /*Wenn ein impliziter cast
existiert. Wird meistens verwendet
(!Verändert die Daten selbst!) */

const_cast /*NUR dazu, const Ref/Zeig die
auf const Daten an Funktionen zu übergeben,
welche nichts const erwarten*/
reinterpret_cast /*Verändert wie Daten an
ihrer  Adresse gelesen („interpretiert“)
werden. (Sehr Hardware nah) */
dynamic_cast /*nicht jede Person ist ein
Mitarbeiter. Mit dynamic_cast zur Laufzeit
prüfen, ob der Cast im Einzelfall erlaubt
ist oder nicht */

int ival = int(pi)  //function-style cast
int ival = (int)pi; //C-Cast
```

operators

scope-operator(::): Mit Hilfe des Scope-Operators ist es möglich in einem anderen Namespace anzusprechen.
(alias Bereichsoperator)

Zugriffsoperator(.): Greift auf einen Member/Methoden in von einem Objekt zu.
Elementverweisoperator(->): Macht das gleiche wie der Zugriffsoperator, mit Pointern
`//(*var).member entspricht also var->member`

Ein-/Ausgabeoperator(>>/<<)

macros

```
#define (@BEZEICHNER)((@repl-list)) (@line)
#define ADD(a, b) ((a) + (b)) /* Rechnung +
Werte in Klammern!!! (vermeidet Fehler) */
```

this Zeiger

Steht für die Speicheradresse einer Instanz mit der eine Methode aufgerufen wird. Wird in Praxis verwendet um Methodenlokale Variablen von Klassenvariablen zu unterscheiden.

modularized class

.h - Headerdatei (Deklaration)

```
#ifndef _(@MODUL)_H_
#define _(@MODUL)_H_
#include <string> //weil im bsp vorhanden

void print((@name) n) { std::cout << n.bsp }

class (@name) {
private:
  //Daten:
  int bsp{0}; //bsp = 0
  std::string txt{“leer“}; //txt = “leer“
public:
  /*Konstruktoren deklarieren:
  + überladene Konst. delegiert*/
  (@name)():(@name)(0, “leer“){};
  (@n)(int _bsp):(@n)(_bsp, “leer“){};
  (@n)(std::string _txt):(@n)(0, _txt){};
  (@n)(int, std::string); //Konstruktor
  //Methoden deklarieren:
  int get_bsp() const; //verbietet schreiben
  void set_Txt(const std::string& s);

  friend void print((@name) n);
  //erlaubt „print“ den Zugriff auf “bsp“
};
#else //auch möglich
#endif
```

.cpp - Quelldatei (Definition)

```
#include <iostream> //in JEDER Quelldatei
#include <string> //weil im bsp vorhanden
#include “(@modul.h)“ //Header inkludieren

/*Konstruktor definieren.
(!nur einen, wegen Delegation!)*/
(@n)::(@n)(int _bsp, std::string _txt)
: (@n)::bsp{ _bsp }, (@n)::txt{ _txt } { };

//Methoden definieren
int (@name)::get_bsp() const {
  return (@name)::bsp;
}

void (@name)::set_Txt(const std::string& s){
  (@name)::bsp = s;
}

//_____________________________________
//Objekt aufrufen (Instanz):
(@name) bezeichner{69, “echt kb mehr -.-“};
```

constructors

```
//Kopierkonstruktor

/*. Automat atm01 = atm02; */
//deklarieren:
(@name)( const (@name) & );
//definieren:
(@name)::(@name)( const (@name) &bez )
   : data1{bez.data1}, … {}
//kopieren verbieten:
(@name)( const (@name) & ) = delete;

//Movekonstruktor
/* Stiehlt Daten aus Objekt.
&&-Argument stellt temporäres Objekt dar.
Parameter dürfen NICHT const sein! */
//deklarieren:
(@name)((@name) && );
//definieren:
(@n)::(@n)((@n) &&bez )
   : data1{bez.data1}, … {
  bez.data1 = /*bsp: 0 oder "" - (leeren)*/;
}

//verschieben verbieten:
(@name)::(@name)((@name) &&bez ) = delete;
```

destructor

Notwendig wenn, Klasse NICHT nur aus Basis und std:: Datentypen besteht. (z.B. pointer)

```
//deklarieren:
~(@name)(); //NIEMALS Parameter angeben!

//definieren:
(@name)::~(@name)() {
  //delete - Anweisungen
}

/* Der Delete-Operator hebt die Zuordnung
des Arbeitsspeichers für ein Objekt auf.
(löst den Destruktor des Objekts aus)*/
delete pointer; //Einzelnes Speicherobjekt
delete[] array; //Array löschen

/*WICHTIG: Objekte die mit new explizit
erstellt wurden müssen auch explizit mit
delete wieder gelöscht werden!*/

/*WICHTIG: Wenn sich der Destruktor in
einer Basisklasse befindet, muss(/sollte!)
er mit virtuel markiert werden!*/
```

Die rule-of-zero besagt, dass wir das Schreiben von benutzerdefinierten Kopier-/Verschiebungskonstruktoren, Zuweisungsoperatoren oder Destruktoren vermeiden können, indem wir vorhandene Typen verwenden, die die entsprechende Kopier-/Verschiebungssemantik unterstützen. (z.B. unique_ptr erzwingt Zuweisungsoperator)

function <- templates -> class

```
template<class T, class T2> //= <typename T>
T bigger(T n1, T n2) {
      if(n1 > n2) { return n1; }
      return n2;
}
//Spezialisierung:
template<> int bigger(int &i1,int &i2) {…}
//Spezialisierung verbieten: = delete;

//dynamic (non-types, types, or templates)
template<class ...T> //variadic templates
```

```
template<class T, class T2> //= <typename T>
struct (@name) {
      T data;
      void set(const T& ob);
};

//Beispiel Methode definieren:
template<class T>
void (@name)<T>::set(const T& obj) {
      data = obj;
}
```

```
/* integral types (bool, char, short, int, long, long long, wchar_t, __wchar_t) können auch
als Template Parameter angegeben werden. Müssen aber zur Compilezeit bekannt sein!*/
template<class T, int n> //Beispiel
```

operator überladen

```
//deklarieren:
class (@name) {
  //…
  //Zuweisungsoperator überladen:
  (@name)& operator=(const (@name)&);
};
//definieren:
(@name)& (@name)::=(const (@name)&) {
  //auf Selbstzuweisung Prüfen!!!
  if(this == &(@name)) { return *this; } … }
//<< & >> benötigen zusätzlich:
friend (@str) operator (@op)((@str), (@name)&);
/* Wobei: <<: (@str):std::istream&
und >>: (@str):std::ostream&   gilt */
```

vererbung/ableiten

```
//class name : [Rechte] Basis, Basis2 {…}
class (@abgel) : public (@Basisklasse) {…};
class (@abgel) : (@Basis1), (@Basis2)  {…};
class (@abgel) : virtual (@Basis) {…};
                 //└=> siehe UML
//Konstruktor definieren:
(@abgel)::(@abgel)(int _bsp)
     : bsp{ _bsp }, (@B)::(@B){ … } { … };
                 //└=> Basis Konstruktor

//Konstruktor wird NICHT implizit vererbt!
using (@B)::(@B); //Explizite Vererbung

/*WICHTIG: Gleichnamige Daten oder Methoden
redefinieren die Werte der Basisklasse
zwar, sind aber mit (@B):: noch aufrufbar.
REGEL: Nur Virtuelle Methoden redefinieren */
//Virtuell(e Methoden)
/* Wenn eine Methode redefiniert werden
soll, wird sie mit virtual gekennzeichnet*/
class (@base) {
  virtual ~(@base)(); //siehe destructor
  …
  //Rein Vituelle Methode mit =0 kennzeichnen
  virtual int add(int x, int y) =0;
};
class (@abgel) : (@base) {
  int add(int x, int y) final {…};
};/*Redefinition mit override/final markieren
W: final-> nicht mehr überschreibbar (weitere
ableitungen) W: Wenn class (@name) final {…}
dann ist (@name) nicht ableitbar* /

/* WICHTIG: Virtuelle Methoden aufrufe im
Konstruktor, rufen die implementierung der
Konstruierten Klasse auf. */
```

Eine abgeleitete Klasse erbt die public-(protected) Datentypen und Methoden der Basis Klasse.
Besteht eine Klasse ausschließlich aus Virtuellen Methoden wird sie als Abstrakte Klasse bezeichnet. (=> KEIN extra Keyword wie interface o.ä. wie in Java notwendig)

enum

```
enum class beispiel{ bsp1, bsp2=5, bsp3 };
                     //  =0    =5     =6
beispiel bsp01{ beispiel::bsp1 };

//impliziter cast seit C++11 nicht möglich!
int bsp02 = beispiel::bsp1; //ERROR
int bsp02{static_cast<int>(beispiel::bsp1)};

//statt int ALLE integral types möglich:
enum class bsp_: bool_{ bsp=true, … };
```

templates: implizite und explizite parameter

```
template<class T> struct Example { … };

//Explizite Template instanzierung:
template struct Example<int>;

//Explizite Paramter spezifizierung:
Example<float> bsp{ 69, … };

//Implizite Argument deduction:
Example bsp{ 6.9 }; //double (C++17)
```

namespace

```
/* Ein namespace erzeugen einen
Gültigkeits-bereich in dem Bezeichner
gelten. */
namespace VIP_Bereich {
      int ivar;
      int funktion();
}
/* Zum definieren, entweder mit scope-
operator zugreifen ( VIP_Bereich::… ),
oder Namespace erneut öffnen
( namespace VIP_Bereich { … } ) */

//namespace alias:
namespace vip = VIP_Bereich;
//einzelne bezeichner einbinden:
using vip::funktion();
//namespace global einbinden:
using namespace vip;

/* ANONYME namespaces können NUR innerhalb
ihrer Quelldatei erreicht werden */
namespace { … } //anonym (ohne Bezeichner)
```

using / typedef

```
//Alias für einen Typ erstellen:
typedef struct Id3_tag Id3_t; // alt
using Id3_t = struct Id3_tag; // C++11
//für Compiler nun: Id3_t = struct Id3_tag

/* WICHTIG: Hierbei handelt es sich um
keinen neu erstellten Typen!
Wird verwendet um Code übersichtlicher
gestallten zu können */
```

iterator

```
/*Iteratoren werden in Containern verwendet um,
durch diese zu interieren, auf Elemente zu
zeigen und Umfang (Start/End) zu definieren */
//Ptr's sind Iterator (Iterator abstrakter Ptr)

/* Folgende Operatoren müssen überladen
werden, damit Klasse ein Iterator ist */
// * zu imitieren
*, -> //um * , wie die Dereferenzierung funzt
[] // array-ähnlichen Zugriff *
+, -, ++, -- //pointer arithmetic *

//for-each Schleife erfordert diese Operatoren:
++i, *i, i!=j
```

storage duration

Code Speicher : Wird den den Arbeitsspeicher geladen. (Executeable Code („text“ segment))

Datenspeicher (Global Data): globale und mit static gekennzeichnete Daten

Stack-Speicher: Im Stack werden Funktionsaufrufe mit ihren **lokalen Variablen** verwaltet

Heap-Speicher: Bei einer Speicheranforderung erhöht sich der Heap Speicher und bei der Freigabe wird er verringert. Einmal reservierter Speicherplatz auf dem Heap, bleibt verfügbar bis er freigegeben wird. Er enthält somit **Array's mit dynamischer Größe** und Objekte mit, **vom Developer festgelegter Lebensdauer** (sprich **new** und **delete**)

atomic operations

Atomic types erlauben einfache Operationen, wie "++" in einer thread-safe Art und Weise (frei von „racing condition“). Ebenfalls können sie für eine lock-free Programmierung verwendet werden – (sehr schwierig / „gefährlich“)
`std::atomic_int`, `std::atomic_uint`, …

inlining

vererbung/ableiten

Besetzt eine Klasse mindestens eine Methode die mit virtual markiert ist, wird sie als polymorphic class bezeichnet.

racing condition

Eine „racing condition“ tritt auf, wenn Threads ein Rennen veranstalten und die daraus resultierende Zeitfolge ein Ergebnis beeinflusst. Beispielsweise wenn zwei Threads mit der selben variable hantieren. (führt zu ungewollten Ergebnissen)

construct order

Reihenfolge in der Objekte Konstruiert werden:
1. Basis Konstruktoren werden aufgerufen (von **links nach rechts**)
2. Nicht statische Attribute werden initialisiert
3. Der eigene Konstrukor Code

(Destruktion in umgekehrter Reihenfolge)

up-, downcast

`instanceof` mit `dynamic_cast` (zur Laufzeit)

```
dynamic_cast<(ptr*)>(…) //error -> nullptr
dynamic_cast<(ref&)>(…) //error -> exception
//benötigt polymorphic classes als Typen!
```

Implizite Casts:
upcast:
Pointer der abgeleiteten Klasse kann Pointer der Base zugeordnet werden.
Objekt der abgeleiteten Klasse kann Objekt der Basis zu geordnet werden.
Das Kopieren eines abgeleiteten Objekts in ein Basisobjekt führt zu “*slicing*” *(vermeiden!)*

downcast:
Basis Objekte können abgeleiteten Klassen *nicht* zugeordnetet werden!

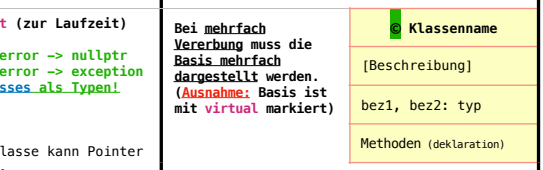
Explizite Casts: `(static/dynamic)_cast<…>(…)`

container access

	vector	deque	list
front			
back			
push_back			
push_front			
pop_front			
at			
operator[]			

UML

Bei mehrfach Vererbung muss die Basis mehrfach dargestellt werden. (Ausnahme: Basis ist mit virtual markiert)



(Erbt aus: —>)(Methode verwendet: - ->)
(Hat als Attribut: <>—)

std:: container

`array`: Array fester Größe (static 1D-C-Array)
`vector`: Array dynam. Größe (dynamic 1D-C-Array)
`deque`: dynam. Array + schneller Zugriff (Anf/End)
`[forward_]list`: [Einfach]/Doppelt Verketteliste
`stack`, `queue`, `priority_queue`: trivial

Wahlweise mit prefix: (“**unordered_**” -> unsortiert. “**multi**” -> erlaubt zusätzlich das mehrfache eintragen von Entrys)
`set`: Zugriff mit frei wählbarem Schlüssel
`map`: Set, aber zwei Elemente statt nur eins

```
//Beispiele:

std::list <int> gqlist1, gqlist2;
for (int i = 0; i < 10; ++i) {
  gqlist1.push_back(i * 2);
  gqlist2.push_front(i * 3);
}

//function for printing the elements in a list
void showlist(std::list <int> g)  {
  std::list <int> :: iterator it;
  for(it = g.begin(); it != g.end(); ++it)
    std::cout << '\t' << *it;
  std::cout << '\n';
}
```

Datatype	Bytes	Type conversions
short	2	Die Konvertierung von v nach T ist beim Casting nach unten oder quer zur Hierarchie nicht immer möglich. Beispielsweise könnte die versuchte Konvertierung mehrdeutig sein, T könnte unzugänglich sein, oder v könnte nicht auf ein Objekt des erforderlichen Typs zeigen (oder sich darauf beziehen).
int	4	
long	4	
long long	8	
char	1	
float	4	
double	8	
long double	12